PARTE 2.A. PREGUNTAS SOBRE LA PRÁCTICA

1. Suponed que en lugar de la implementación del TAD Pila elaborado para la práctica 2 con memoria estática, se desea hacer una implementación basada en listas. Indicad cómo influye el uso de esta nueva implementación desde los puntos de vista sintáctico, semántico y uso de memoria.

El código de las primitivas del TAD Pila se simplificaría mucho porque simplemente llamarían a las primitivas del TAD Lista. Por ejemplo, tanto insertar como extraer un elemento de la pila, llamarían a las funciones que insertan y extraen un nodo por el principio de la lista.

2. Suponed que se disponen de dos implementaciones diferentes del TAD lista (no ordenada). La primera de ellas es la que se ha elaborado en la presente práctica con memoria dinámica. La segunda es una implementación con memoria estática, donde se emplea un array para almacenar las direcciones a los elementos que contienen la lista, y una referencia (puntero) a la posición de cabeza de la misma.

Si se desea eliminar el elemento que se encuentra justo en la mitad de la lista, indique los pasos que debe realizarse en cada caso y calcule el número de accesos a memoria en cada uno de los mismos.

Si esta operación tuviera que repetirse con mucha asiduidad, ¿por cuál de las dos implementaciones optaría?

PRIMER CASO:

1. Recorremos la lista para saber cuántos elementos contiene.
2. Situamos un puntero auxiliar en el nodo anterior al que está en la mitad (nodo 1).
3. En otro puntero auxiliar guardamos la dirección del nodo que está en la mitad (nodo 2).
4. En un elemento copiamos el campo info del nodo 2.
5. Ajustamos el campo next del nodo 1 al siguiente del nodo 2.
6. Liberamos el nodo 2 mediante el segundo puntero auxiliar.
7. Habría que tener en cuenta las condiciones de error: si la lista está vacía o si tiene un elemento.

SEGUNDO CASO:

1. Recorremos el array hasta que el puntero a elemento es nulo.
2. Situamos el puntero auxiliar en el nodo que está en la mitad (nodo 1).
3. Con un bucle, liberamos el nodo al que apunta el auxiliar y copiamos el nodo siguiente en esa posición, hasta que el nodo siguiente sea nulo.
4. Al salir del bucle, se libera el último nodo y se hace que apunte a NULL.
5. Habría que tener en cuenta las condiciones de error: si la lista está vacía.

Optaríamos por la primera implementación porque es mucho más eficiente. En cuanto llega al nodo que se quiere liberar, se hacen una serie de pasos inmediatos. Mientras que de la otra forma necesita un bucle para recorrer la segunda mitad del array e ir copiando y liberando los nodos.

PARTE 2.B. MEMORIA SOBRE LA PRÁCTICA

1. Decisiones de diseño

En el ejercicio que más decisiones tuvimos que tomar fue en el quinto ejercicio, al realizar el TAD Solution. La estructura de tipo solution tenía que almacenar el array de tipo Move con el que se ha encontrado el camino, el TAD usado y el número de movimientos del camino. Al final, no guardamos el camino en sí, porque en ningún momento del programa se utiliza. Al final, decidimos darle a todos los elementos de la estructura memoria estática, simplemente por motivo de simplificar el código y reducir el número de mallocs, que complican y dificultan la programación. A la hora de decidir qué funciones declarar en este TAD, nos basamos en el resto de TADs. Definimos una función que inicializa y libera una solución, otras que insertan y extraen información en una solución, otra que copia una solución y una última que lo imprime. En el resto de ejercicios no tuvimos que tomar decisiones tan importantes, dado que la declaración de las funciones venía dada por el enunciado. Únicamente, decidimos cambiar lo que devuelve alguna función por Status, porque en caso de error, se perdería memoria.

2. Informe de uso de memoria

Capturas de pantalla con las salidas del análisis de la herramienta memcheck de Valgrind:

Ejercicio 1:

```
==1717== HEAP SUMMARY:
==1717==     in use at exit: 0 bytes in 0 blocks
==1717==   total heap usage: 25 allocs, 25 frees, 99,280 bytes allocated
==1717==
==1717== All heap blocks were freed -- no leaks are possible
==1717==
==1717== For counts of detected and suppressed errors, rerun with: -v
==1717== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Ejercicio 2:

```
==2821== HEAP SUMMARY:
==2821==     in use at exit: 0 bytes in 0 blocks
==2821==   total heap usage: 381,593 allocs, 381,593 frees, 6,283,496 bytes allocated
==2821==
==2821== All heap blocks were freed -- no leaks are possible
==2821==
==2821== For counts of detected and suppressed errors, rerun with: -v
==2821== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Ejercicio 3:

```
==3398== HEAP SUMMARY:
==3398==     in use at exit: 0 bytes in 0 blocks
==3398==   total heap usage: 1,160 allocs, 1,160 frees, 812,096 bytes allocated
==3398==
==3398== All heap blocks were freed -- no leaks are possible
==3398==
==3398== For counts of detected and suppressed errors, rerun with: -v
==3398== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Ejercicio 4:

```
==3450== HEAP SUMMARY:
==3450==     in use at exit: 0 bytes in 0 blocks
==3450==   total heap usage: 123 allocs, 123 frees, 1,544 bytes allocated
==3450==
==3450== All heap blocks were freed -- no leaks are possible
==3450==
==3450== For counts of detected and suppressed errors, rerun with: -v
==3450== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Ejercicio 5:

```
==3678== HEAP SUMMARY:
==3678==     in use at exit: 0 bytes in 0 blocks
==3678==   total heap usage: 2,421 allocs, 2,421 frees, 1,626,120 bytes allocated
==3678==
==3678== All heap blocks were freed -- no leaks are possible
==3678==
==3678== For counts of detected and suppressed errors, rerun with: -v
==3678== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Como muestran las capturas de pantalla, no se pierde memoria ni se produce ningún error sobre el uso inapropiado de la memoria en ningún ejercicio.

3. Conclusiones finales

Estos cuatro ejercicios nos han sido muy útiles para reforzar lo visto en clase sobre pilas y colas y para poner en práctica la teoría más reciente sobre listas. Los ejercicios en los que más dificultades encontramos fueron el tercero y el quinto, dado que eran más diferentes a los que ya habíamos hecho en anteriores prácticas. El primer ejercicio no tardamos mucho en hacerlo y nos sirvió para asentar los conocimientos sobre colas. El segundo ejercicio es muy parecido al tercer ejercicio de la anterior práctica, salvo usando colas en vez de pilas, así que solo modificamos las llamadas a funciones definidas en el TAD Pila y las cambiamos por sus simétricas en el TAD Cola. Para imprimir el camino usando colas, necesitamos realizar una modificación mayor. Al guardar el camino en una cola, se guarda al revés (desde la salida hacia la entrada). Para imprimirlo correctamente, teníamos que darle la vuelta a la cola. Para ello, necesitamos de dos colas auxiliares, de manera que se extrae el primer elemento de la cola con el recorrido y se inserta en una auxiliar. Y al extraer el siguiente, traspasamos todo el contenido de la primera en la segunda, se inserta el elemento extraído en la primera y se vuelve a traspasar la segunda auxiliar en la primera. En el tercer ejercicio nos encontramos más dificultades porque era el más diferente al resto. Al principio, nos costó entender el enunciado, dado que no dejaba clara la salida esperada. Después, juntamos el algoritmo que indica si hay camino posible usando una pila (p2_e3.c que ahora será la función mazesolver_stack) con el que usa una cola (p3_e2.c que ahora será la función mazesolver_queue). Finalmente, hicimos una función main en el propio solver_mazes.c para comprobar el correcto funcionamiento de las nuevas funciones creadas. Esta función declara un array de arrays de tamaño 4 de tipo Move y después llama a la función mazesolver_run. Esta, simplemente llama a las dos funciones anteriormente mencionadas e imprime por pantalla si hay o no camino posible según el TAD usado y el número de movimientos de dicho camino. El ejercicio cuatro fue más sencillo y nos sirvió como toma de contacto con el TAD Lista que acabábamos de ver en teoría. Finalmente, el ejercicio cinco fue sin duda el más costoso, dado que unía todos los TADs vistos hasta esa fecha. Fue el que más tardamos en hacer porque teníamos que hacer el TAD solución al completo y tuvimos que decidir que estructura tendría, como se ha explicado en el apartado de diseño.